Session - 9 : Advanced Cypress Techniques

Duration: 3 hours

Handling Frames in Cypress

This lecture covers handling frames in Cypress, including the necessary setup and code implementation.

Key Points

1. **Frames Overview**:
    - Frames are HTML documents embedded within another HTML document.
    - To interact with elements inside a frame, you must switch the context to that frame.
2. **Cypress Frame Support**:
    - Earlier versions of Cypress did not support frames, but recent versions do.
    - You need to install and configure the `cypress-iframe` plugin to work with frames in Cypress.

Steps to Handle Frames in Cypress

1. **Install `cypress-iframe` Plugin**:
    - Run the following command to install the plugin:

```
shCopy code
npm install -D cypress-iframe
```

2. **Import `cypress-iframe` Plugin**:
    - Import the plugin at the top of your test file:

```
javascriptCopy code
import 'cypress-iframe';
```

3. **Load and Switch to Frame**:
    - Use `cy.frameLoaded()` to load the frame into the Cypress object.
    - Use `cy.iframe()` to switch the context to the frame.

Example Code

Setting Up the Test

1. **Visit the Page and Load the Frame**:
    - First, visit the page containing the frame.
    - Use the frame ID to load the frame.

```
javascriptCopy code
// Visit the page
cy.visit('https://rahulshettyacademy.com/AutomationPrac
tice/');

// Load the frame
cy.frameLoaded('#courses-iframe');
```

2. **Switch to Frame and Perform Actions**:
    - Switch to the frame and perform actions like clicking a button or validating elements.

```
javascriptCopy code
// Switch to the frame and click the 'Mentorship' link
```

```
cy.iframe().find('a[href*="mentorship"]').first().click
();

// Validate the number of packages available in the 'Me
ntorship' section
cy.iframe().find('h1[class*="pricing-title"]').should
('have.length', 2);
```

Full Test Code

Here's a complete example integrating all the steps:

```
import 'cypress-iframe'
describe('template spec', () => {
    it('passes', () => {
      cy.visit('https://rahulshettyacademy.com/AutomationPrac
tice/')
    //   cy.get('div.mouse-hover-content').invoke('show')
      cy.frameLoaded('#courses-iframe')
      cy.iframe().find('a[href*="mentorship"]').eq(0).click()

})
})
```

Explanation

1. **Loading the Frame**:
   - `cy.frameLoaded('#courses-iframe')` ensures Cypress recognizes the frame and loads it into the Cypress object.
2. **Switching Context**:
   - `cy.iframe()` switches the context to the frame, allowing interaction with elements inside the frame.
3. **Interacting with Elements**:

- `cy.iframe().find('a[href*="mentorship"]').first().click()` clicks the 'Mentorship' link inside the frame.
- `cy.iframe().find('h1[class*="pricing-title"]').should('have.length', 2)` asserts that there are two elements with the class `pricing-title` inside the frame.

This approach ensures that Cypress can handle and interact with elements inside frames effectively.

Understanding Test Hooks in Cypress

Cypress provides several test hooks, inherited from the Mocha testing framework, which allow you to set conditions that run before and after your tests. These hooks are similar to annotations in TestNG if you are familiar with Selenium.

- `before` **Hook**: Runs once before all tests in the block. Use this to initialize test data, invoke the browser, or hit the URL.
- `after` **Hook**: Runs once after all tests in the block. Use this for cleanup tasks such as deleting test data.
- `beforeEach` **Hook**: Runs before each individual test case in the block.
- `afterEach` **Hook**: Runs after each individual test case in the block.

The order of execution is:

1. `before`
2. `beforeEach`
3. Test case (`it` block)
4. `afterEach`
5. `after`

This order ensures proper setup and teardown for your tests.

Writing a Sample Test

Let's create a new test file and use the `before` hook to set up our test environment. Here's how you can write a basic test to enter a name and select a gender on a sample e-commerce application.

HTML Structure:

```
htmlCopy code
<input name="name" class="form-control" />
<select class="form-control">
  <option value="male">Male</option>
  <option value="female">Female</option>
</select>
```

Cypress Test:

```
describe('E-commerce Application', function(){
  before(function() {
    // Setup code goes here
    cy.fixture('example').then(function(data){
    this.data=data
    })
  })

  it('should enter name and select gender', function(){
    cy.visit('https://rahulshettyacademy.com/angularpractic
e/'); // Replace with actual URL
    cy.get('input[name="name"]:nth-child(2)').type(this.data.
name);
    cy.wait(7000)
    cy.get('#exampleFormControlSelect1').select(this.data.gen
der);
  })

  after(() => {
    // Teardown code goes here
  })
})
```

Driving Data from External Sources in Cypress

This lecture explains how to use external data sources in Cypress by utilizing JSON files stored in the `fixtures` folder. This allows for easy data management and reuse across multiple test cases.

Key Points

1. **Fixtures in Cypress**:
   - Fixtures are external files used to manage test data.
   - Cypress interacts with JSON files placed in the `fixtures` folder.
2. **Creating a JSON File**:
   - A sample JSON file (`example.json`) is created in the `fixtures` folder:

```
jsonCopy code
{
  "name": "Bob",
  "gender": "male"
}
```

3. **Loading Fixtures in Cypress**:
   - Use the `cy.fixture()` method to load the JSON file.
   - It's recommended to load fixtures in a `before` hook to separate setup logic from test logic.

Steps to Use Fixtures in Cypress

1. **Create a JSON File**:
   - Create a JSON file in the `fixtures` folder (e.g., `example.json`):

```
jsonCopy code
{
  "name": "Bob",
  "gender": "male"
```

```
}
```

2. **Load the JSON File in a `before` Hook**:
   - Load the fixture file using `cy.fixture()` in a `before` hook.
   - Resolve the promise and assign the data to a global variable using `this`.

Example Code

Test Setup with Fixtures

1. **Loading Fixture Data in the `before` Hook**:
   - Load the fixture data and assign it to a global variable.

```
describe('E-commerce Application', function(){
  before(function() {
    // Setup code goes here
    cy.fixture('example').then(function(data){
    this.data=data
    })
  })

  it('should enter name and select gender', function(){
    cy.visit('https://rahulshettyacademy.com/angularpra
ctice/'); // Replace with actual URL
    cy.get('input[name="name"]:nth-child(2)').type(thi
s.data.name);
    cy.wait(7000)
    cy.get('#exampleFormControlSelect1').select(this.da
ta.gender);
  })

  after(() => {
    // Teardown code goes here
  })
```

```
})
```

Explanation

1. **Loading the Fixture File**:
   - `cy.fixture('example')` loads the `example.json` file from the `fixtures` folder.
   - The `.then(function(data) {...})` block resolves the promise and assigns the data to `this.data`.
2. **Using the Fixture Data in Tests**:
   - `this.data` contains the entire JSON object.
   - Access specific values using `this.data.name` and `this.data.gender`.

Full Test Code

Here's the complete example integrating all the steps:

```
javascriptCopy code
describe('Fixture Data Test', () => {
  before(function() {
    // Load the fixture file
    cy.fixture('example').then(function(data) {
      this.data = data; // Assign data to a global variable
    });
  });

  it('should use data from fixture', function() {
    // Access the fixture data using 'this.data'
    cy.log(this.data.name); // Output: Bob
    cy.log(this.data.gender); // Output: male

    // Use the data in your test
    cy.visit('your-test-url');
    cy.get('input[name="name"]').type(this.data.name);
```

```
    cy.get('input[name="gender"]').type(this.data.gender);
  });
});
```

Summary

- **Fixtures**: Use the `fixtures` folder to store external data.
- **Loading Fixtures**: Use `cy.fixture()` to load and resolve JSON data.
- **Using Data**: Assign the data to a global variable with `this` for easy access in tests.
- **Test Separation**: Use a `before` hook to keep test setup logic separate from test execution.

This approach makes it easy to manage and reuse test data, leading to more maintainable and readable tests.

Implementing Test Automation in Cypress

In this lecture, the focus is on automating some interesting scenarios using Cypress and verifying different aspects of an input field and a disabled radio button.

Key Scenarios

1. **Two-Way Data Binding**:
    - Enter a value in an edit box and verify if it appears in another edit box.
    - Verify the input field has a minimum length attribute of 2.
    - Verify if a specific element (e.g., a radio button) is disabled.

Detailed Steps

Scenario 1: Two-Way Data Binding Verification

1. **Input a Value and Verify Two-Way Binding**:

- Enter a value in the first input field.
- Verify that the same value appears in the two-way bound input field.

2. **Validation for Minimum Length**:
   - Check if the `minlength` attribute of the input field is set to 2.
3. **Verify Element is Disabled**:
   - Verify if a radio button or any other element is disabled.

Example Code

Cypress Test Implementation

1. **Test for Two-Way Data Binding**:
   - Input a value and check if it is reflected in another field.
   - Validate the `minlength` attribute.
   - Check if the element is disabled.

```
javascriptCopy code
describe('Form Validation Tests', () => {
  before(function() {
    // Load fixture data if needed
    cy.fixture('example').then(function(data) {
      this.data = data; // Assign data to a global variable
    });
  });

  it('should validate two-way data binding, minlength, and di
sabled state', function() {
    // Visit the URL
    cy.visit('your-test-url');

    // Enter value in the input field
    cy.get('input[name="name"]').type(this.data.name);
```

```
    // Verify two-way data binding
    cy.get('input[name="name"]').should('have.value', this.da
ta.name);

    // Validate minlength attribute
    cy.get('input[name="name"]').should('have.attr', 'minleng
th', '2');

    // Verify if the radio button is disabled
    cy.get('#radio-button-id').should('be.disabled');
  });
});
```

Explanation

1. **Two-Way Data Binding Verification**:
   - Use `cy.get('input[name="name"]')` to target the input field.
   - Use `.type(this.data.name)` to enter the value.
   - Use `.should('have.value', this.data.name)` to check the value in the two-way bound input field.
2. **Minlength Attribute Validation**:
   - Use `.should('have.attr', 'minlength', '2')` to verify the `minlength` attribute.
3. **Disabled State Verification**:
   - Use `.should('be.disabled')` to verify if the radio button is disabled.

Full Test Code

Here's the complete example integrating all the steps:

```
javascriptCopy code
describe('Form Validation Tests', () => {
  before(function() {
    // Load fixture data if needed
```

```
    cy.fixture('example').then(function(data) {
      this.data = data; // Assign data to a global variable
    });
  });

  it('should validate two-way data binding, minlength, and di
sabled state', function() {
    // Visit the URL
    cy.visit('your-test-url');

    // Enter value in the input field
    cy.get('input[name="name"]').type(this.data.name);

    // Verify two-way data binding
    cy.get('input[name="name"]').should('have.value', this.da
ta.name);

    // Validate minlength attribute
    cy.get('input[name="name"]').should('have.attr', 'minleng
th', '2');

    // Verify if the radio button is disabled
    cy.get('#radio-button-id').should('be.disabled');
  });
});
```

Summary

- **Two-Way Data Binding**: Ensure input values reflect correctly across bound elements.
- **Attribute Validation**: Validate specific attributes like `minlength` directly.
- **State Verification**: Check if elements are disabled using `should('be.disabled')`.

This approach ensures comprehensive validation of input fields, attributes, and states, which is crucial for robust test automation.

Implementing a Generic Add to Cart Function in Cypress

In this lecture, the focus is on creating a generic function in Cypress to add products to the cart based on their names. This involves navigating to a product page, finding products by their names, and clicking the corresponding "Add" buttons. Additionally, we'll create a custom Cypress command to make this functionality reusable and dynamic.

Key Scenarios

1. **Navigate to the Products Page**:
    - Click on the "Shop" button to navigate to the products page.
2. **Select and Add Products**:
    - Create a function that accepts a product name, finds the corresponding product on the page, and clicks the "Add" button.
    - Extend this function to handle multiple product names, adding each to the cart.
3. **Create a Custom Cypress Command**:
    - Encapsulate the logic for adding products to the cart in a custom Cypress command for reusability and clarity.

Detailed Steps

Step 1: Navigating to the Products Page

First, we need to navigate to the products page by clicking the "Shop" button. This can be done using a CSS selector to locate and click the button.

```
javascriptCopy code
cy.get('button#shop').click();
```

Step 2: Selecting and Adding Products to the Cart

Next, we will create a function to find the product based on its name and click the corresponding "Add" button. We will use Cypress commands to grab all product titles, iterate through them, and click the "Add" button for the matching product.

```
javascriptCopy code
cy.get('.product-title')  // Locator for all product titles
  .each(($el, index, $list) => {
    if ($el.text().includes(productName)) {
      cy.get('.add-button').eq(index).click();  // Locator fo
r all add buttons
    }
  });
```

Step 3: Creating a Custom Cypress Command

To make our function reusable, we will create a custom Cypress command. This command will encapsulate the logic of selecting and adding a product to the cart.

1. **Define the Custom Command**:
   - We will define our custom command in the `commands.js` file located in the `support` directory.
2. **Implementation**:
   - The custom command will accept a product name as an argument, find the product, and click the "Add" button.

```
javascriptCopy code
// In commands.js

Cypress.Commands.add('selectProduct', (productName) => {
  cy.get('.product-title')  // Locator for all product titles
    .each(($el, index, $list) => {
      if ($el.text().includes(productName)) {
        cy.get('.add-button').eq(index).click();  // Locator
```

```
for all add buttons
      }
    });
});
```

Step 4: Using the Custom Command in Tests

Now, we can use our custom command in Cypress tests to add products to the cart dynamically.

```
javascriptCopy code
describe('Add Products to Cart', () => {
  it('should add specified products to the cart', () => {
    // Visit the shop page
    cy.visit('your-test-url');

    // Click on the shop button to navigate to products page
    cy.get('button#shop').click();

    // Add products to the cart
    cy.selectProduct('BlackBerry');
    cy.selectProduct('Nokia Edge');

    // Verify products are in the cart (implementation depend
s on your app)
    cy.get('.cart-item').should('contain', 'BlackBerry');
    cy.get('.cart-item').should('contain', 'Nokia Edge');
  });
});
```

Explanation

1. **Navigate to Products Page**:

- `cy.get('button#shop').click();` clicks the shop button to navigate to the products page.

2. **Selecting and Adding Products**:
   - `cy.get('.product-title').each(($el, index, $list) => { ... })` iterates through each product title.
   - If the product title matches the specified name, `cy.get('.add-button').eq(index).click();` clicks the corresponding "Add" button.
3. **Creating a Custom Cypress Command**:
   - `Cypress.Commands.add('selectProduct', (productName) => { ... })` defines the custom command.
   - The command encapsulates the logic for finding and adding the specified product to the cart.
4. **Using the Custom Command in Tests**:
   - `cy.selectProduct('BlackBerry');` uses the custom command to add "BlackBerry" to the cart.
   - `cy.get('.cart-item').should('contain', 'BlackBerry');` verifies the product is in the cart.

Summary

- **Navigation**: Click the shop button to navigate to the products page.
- **Product Selection**: Find and add products based on their names.
- **Custom Command**: Create a reusable custom command to encapsulate the product selection logic.
- **Test Integration**: Use the custom command in tests to dynamically add products to the cart and verify their presence.

Enhancing Cypress Tests with Custom Commands and Data-Driven Approaches

In this follow-up, we'll enhance our Cypress test by integrating custom commands and driving data from a fixture file. This approach ensures reusability and maintains clean, organized test code.

Goals

1. **Use Custom Commands**:
   - Reuse the logic for adding products to the cart using a custom command.
2. **Data-Driven Tests**:
   - Read product names from a fixture file (`example.json`) and iterate through them to add each product to the cart.

Detailed Steps

Step 1: Creating a Custom Command

We'll start by defining a custom command in `commands.js` to encapsulate the logic of adding a product to the cart.

commands.js:

```
javascriptCopy code
Cypress.Commands.add('selectProduct', (productName) => {
  cy.get('.product-title')
    .each(($el, index) => {
      if ($el.text().includes(productName)) {
        cy.get('.add-button').eq(index).click();
      }
    });
});
```

Step 2: Defining the Fixture Data

Next, we'll create a fixture file (`example.json`) containing the product names to be added to the cart.

example.json:

```
jsonCopy code
{
  "products": ["BlackBerry", "Nokia Edge"]
}
```

Step 3: Writing the Test Case

Now, we write our Cypress test case, which reads the product names from the fixture file and uses the custom command to add each product to the cart.

test.js:

```
javascriptCopy code
describe('Add Products to Cart', () => {
  before(() => {
    // Load fixture data
    cy.fixture('example').then((data) => {
      cy.wrap(data).as('testData');
    });
  });

  it('should add specified products to the cart', function()
{
    // Visit the shop page
    cy.visit('your-test-url');

    // Click on the shop button to navigate to products page
    cy.get('button#shop').click();

    // Iterate through products array and add each product to
the cart
    this.testData.products.forEach((product) => {
      cy.selectProduct(product);
    });
```

```
    // Verify products are in the cart
    this.testData.products.forEach((product) => {
      cy.get('.cart-item').should('contain', product);
    });
  });
});
```

Explanation

1. **Custom Command**:
   - The custom command `selectProduct` takes a product name as an argument, finds the product on the page, and clicks the "Add" button.
2. **Fixture File**:
   - The fixture file `example.json` contains an array of product names to be added to the cart.
3. **Test Case**:
   - **Before Hook**: Loads the fixture data before running the test.
   - **Test Execution**:
     - Visits the shop page and clicks the "Shop" button.
     - Iterates through the `products` array from the fixture file and calls the custom command for each product.
     - Verifies that each product is added to the cart.

Enhancing the Test

Handling Case Sensitivity

We encountered an issue where the product name's case affected the selection. We can address this by normalizing the text comparison to be case-insensitive.

commands.js (Updated):

```
javascriptCopy code
Cypress.Commands.add('selectProduct', (productName) => {
  cy.get('.product-title')
    .each(($el, index) => {
      if ($el.text().toLowerCase().includes(productName.toLow
erCase())) {
        cy.get('.add-button').eq(index).click();
      }
    });
});
```

Optimizing with Data-Driven Tests

To handle multiple products more efficiently, we'll read the product names from a fixture file and use the `forEach` method to iterate through them.

test.js (Updated):

```
javascriptCopy code
describe('Add Products to Cart', () => {
  before(() => {
    // Load fixture data
    cy.fixture('example').then((data) => {
      cy.wrap(data).as('testData');
    });
  });

  it('should add specified products to the cart', function()
{
    // Visit the shop page
    cy.visit('your-test-url');

    // Click on the shop button to navigate to products page
    cy.get('button#shop').click();
```

```
    // Iterate through products array and add each product to
the cart
    this.testData.products.forEach((product) => {
      cy.selectProduct(product);
    });

    // Verify products are in the cart
    this.testData.products.forEach((product) => {
      cy.get('.cart-item').should('contain', product);
    });
  });
});
```

Summary

1. **Custom Commands**:
   - Encapsulated the logic for adding a product to the cart in a reusable custom command.
2. **Data-Driven Tests**:
   - Utilized fixture files to manage test data and drive the test execution dynamically.
3. **Optimized Test Execution**:
   - Implemented an efficient, maintainable test that handles multiple products seamlessly.

By following these steps, you can ensure your Cypress tests are modular, reusable, and data-driven, making them easier to maintain and extend.

Debugging Cypress Tests: An Overview

Debugging is a crucial aspect of test automation, helping you understand and fix issues in your tests. Cypress offers several powerful features for debugging. Here's a summary of how to effectively use these features:

Key Debugging Features in Cypress

1. **Pausing Test Execution**

   Cypress allows you to pause your test execution at any point. This can be useful if you want to investigate the application state or inspect elements.

   Usage:

   ```
   javascriptCopy code
   cy.pause();
   ```

   Example:
   If you want to pause the test after a specific step:

   ```
   javascriptCopy code
   cy.get('button#shop').click();
   cy.pause(); // Pauses here
   cy.get('.cart').should('contain', 'BlackBerry');
   ```

   How It Works:

   - When you use `cy.pause()`, Cypress stops executing the test at that point.
   - You can inspect the application, view the current state, and interact with it manually.
   - Once you've resolved the issue, you can resume the test by clicking the "Resume" button in the Cypress Test Runner.

2. **Time Travel**

   Cypress provides a time-travel feature that allows you to view screenshots of your application at each step of the test execution. This helps you understand how your application's state changes over time.

Usage:

- Navigate through your test steps in the Cypress Test Runner to view screenshots and the application's state at each step.

3. **Console Logs**

   Cypress logs detailed information about each command and assertion in the browser's console. You can access these logs to see what Cypress did at each step and diagnose issues.

   How to Access:

   - Open the browser's Developer Tools (usually by pressing `F12` or `Ctrl+Shift+I`).
   - Go to the "Console" tab to view logs generated by both your application and Cypress.

4. **Debug Command**

   Cypress also provides a `debug()` command, which is similar to `pause()` but used in a more specific context. It can be attached to any Cypress command to stop execution and open the browser's DevTools at that point.

   Usage:

   ```
   javascriptCopy code
   cy.get('button#shop').click().debug();
   ```

   How It Works:

   - When Cypress reaches a `debug()` command, it pauses the test and opens the DevTools.
   - This allows you to inspect the state and interact with the application as you would with `pause()` but in the DevTools environment.

Example of Debugging a Test

test.js:

```
javascriptCopy code
describe('Add Products to Cart', () => {
  before(() => {
    cy.fixture('example').then((data) => {
      cy.wrap(data).as('testData');
    });
  });

  it('should add specified products to the cart', function()
{
    cy.visit('your-test-url');
    cy.get('button#shop').click();

    // Adding debug here
    this.testData.products.forEach((product) => {
      cy.selectProduct(product).debug(); // Stops here and op
ens DevTools
    });

    // Verify products are in the cart
    this.testData.products.forEach((product) => {
      cy.get('.cart-item').should('contain', product);
    });
  });
});
```

Best Practices for Debugging

1. **Use `cy.pause()` and `debug()` Wisely**:
   - Only use `cy.pause()` and `debug()` during development. Remember to remove them from your final test code to avoid unintended pauses.
2. **Inspect Screenshots and Logs**:

- Utilize Cypress’s time-travel feature and console logs to diagnose issues effectively.

3. **Review Cypress Documentation**:
    - For detailed information on debugging and other features, refer to the Cypress documentation.

Next Steps

- **Page Object Design Pattern**: In the upcoming lectures, we will explore how to use the Page Object Design Pattern to further optimize our tests.
- **Additional Scenarios**: We'll also look at validating more complex scenarios, such as verifying totals in the cart.

By leveraging these debugging features, you can efficiently identify and fix issues in your Cypress tests, leading to more reliable and maintainable test cases.